Cameron Zurmuhl

CS203 Computer Organization

Professor Pfaffmann

22 October 2017

Project 1 Report

Introduction:

Project 1 focused on making a tool chain to simulate how a computer interprets instructions. The project is composed of a three-program tool chain which interprets instructions, simulates a CPU process in a GUI, and examines memory at arbitrary locations in a GUI. In my report, I will discuss the design and implementation for each program in the tool chain.

Program 1: Assembler

The assembler program reads in LEGv8 assembly code and parses lines to hexadecimal instructions. Those instructions are saved in a main memory class, which is written to a ".o" image file at the end of the assembler execution. Below I will show my ISA for the instructions that we had to implement. Then, I will describe the implementation of the assembler.

Little-Finger ISA

B Type

Instruction	Opcode
B	0x00A0
BL	0x04A0

CB Type

B.cond	0x2A0
CBNZ	0x5A8
CBZ	0x5A0

R Type

ADD	0x0458
ADDS	0x0558
SUB	0x0658
SUBS	0x0758
AND	0x0450
ORR	0x0550
EOR	0x0650
BR	0x06B0
LSL	0x069B
LSR	0x069A

I Type

ADDI	0x0448
ADDIS	0x0588
SUBI	0x0688
SUBIS	0x0788
ANDI	0x0490
ORRI	0x0590
EORI	0x0690

D Type

LDUR	0x7C2
STUR	0x7C0
LDURSW	0x5C4
STURW	0x5C0
LDURH	0x3C2
STURH	0x3C0
LDURB	0x1C2
STURB	0x1C0

Stack Type

PUSH (add memory to stack data)	0x0222
POP (take data from stack data)	0x0333

*****Stack and Frame Pointers are updated

Special Type

NOP	0x00000000
HALT	0x11111111

MOVEZ

MOVEZ	0x694

Instruction Sizes: 32 bits. All operations in Verilog are transferred from the green card unless specified. All instructions are in hexadecimal

B-Type: 16-bit opcode, 16-bit BR address.

Special-Type: 32-bit opcode

Stack-Type: 16-bit opcode, 16-bit Rd

PUSH: Memory[stack]= Rd

POP: Rd = Memory[stack];

R-Type: 16-bit opcode, 4-bit Rm, 4-bit shamt, 4-bit Rn, 4-bit Rd

D-Type: 12-bit opcode, 12-bit DT_address, 4 bit-Rn, 4-bit Rt

MOVEZ: 12-bit opcode, 4-bit Rd, 16-bit address

Rd = Memory[address]

I-Type: 16-bit opcode, 8-bit immediate, 4-bit Rn, 4-bit Rd

CB-Type: 12-bit opcode, 4-bit Rt, 16-bit Cond_BR_Address

Implementation of assembler:

To implement the assembler first I had to read in lines to parse. The data structure of choice was a Queue, implemented with a LinkedList (1,2). The dynamic structure allowed an easy and efficient flow of lines to parse. I stripped the lines of all irrelevant information and continued with a one-pass reading. Every directive that was parsed was stored in a HashMap(3) with their relative positions in memory. The HashMap came in use with the line “MOVEZ X0, data." In the assembly code, data is not defined before we reach the line. The solution was trivial: enqueue the line again so data can be reached and define. Store "MOVEZ" as a key in the HashMap with its relative position as the value. When "MOVEZ" is encountered again, it can be parsed successfully and loaded to memory in its proper location obtained from the HashMap.

Once all the lines are parsed into hexadecimal instructions and loaded into Byte-addressable memory, the assembler writes an image file which contains the memory printed Byte-by-Byte and an image on the first line of the machine. The information stored in the image is the register count, max memory, and location of Frame and Stack Pointer. Once the assembler completed execution, summary statistics are printed to the command line.

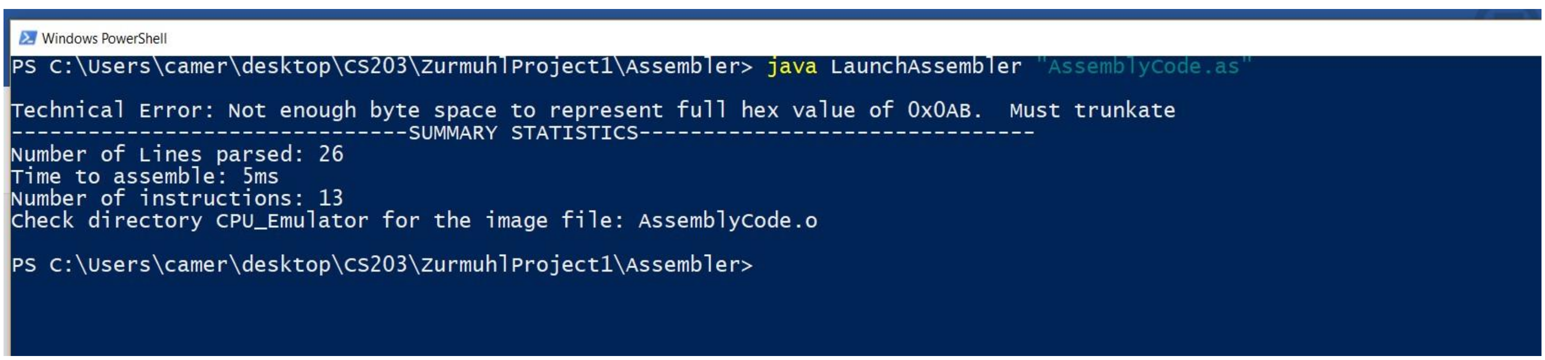


Figure 1: Results of Assembler Program

The technical error that was thrown was because I tried to store the hex value 0x0AB in a 1-byte data field. So, I had to drop the 0 in the hex, which doesn't change the result of the storage. If the hex value was 0xAAB, on the other hand, only AB will be stored, and the user would have to change the data field size. Other than the error, summary statistics are shown accordingly. Upon inspection, the file "AssemblyCode.o" was successfully uploaded.

Program 2: CPU Simulator-Design/Implementation

The CPU Simulator is the main program that drives the project and has the most user functionality. The backend of the simulator involves a fetch-decode-execute-update cycle until all the instructions are processed in memory. Fetching involves obtaining a four-byte instruction from a memory address, which is specified by the Program Counter, and loading that instruction into the Instruction Register. Decoding the instruction involves analyzing the opcode of the instruction, and returning its instruction type via reversed parsing. Once the instruction name is known by interpreting its type and searching for the matching opcode held in an array, method execute performs the instruction and all necessary operations on the registers/memory. Finally, update resets any flags that were set during the process. The process repeats when the Program Counter is set to the next instruction (it advances four bytes in memory). The branching performed in the assembly code is either position specific or PC-relative, so all fetch cycles can rely on the Program Counter.

The Simulator constructor itself accepts a file name (to decode) and a Boolean value on whether to operate on noisy mode. Noisy mode prints out every instruction interpreted and what happens to the registers/memory upon execution. These commands are wrapped into a GUI which the user can control:

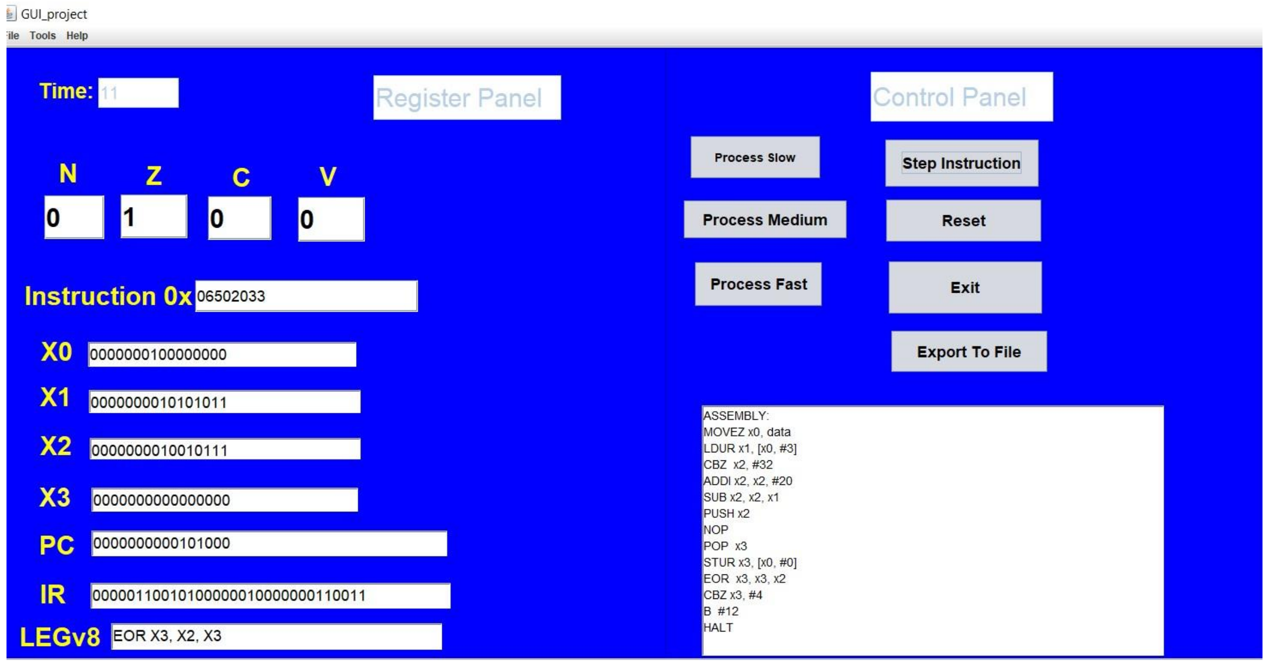
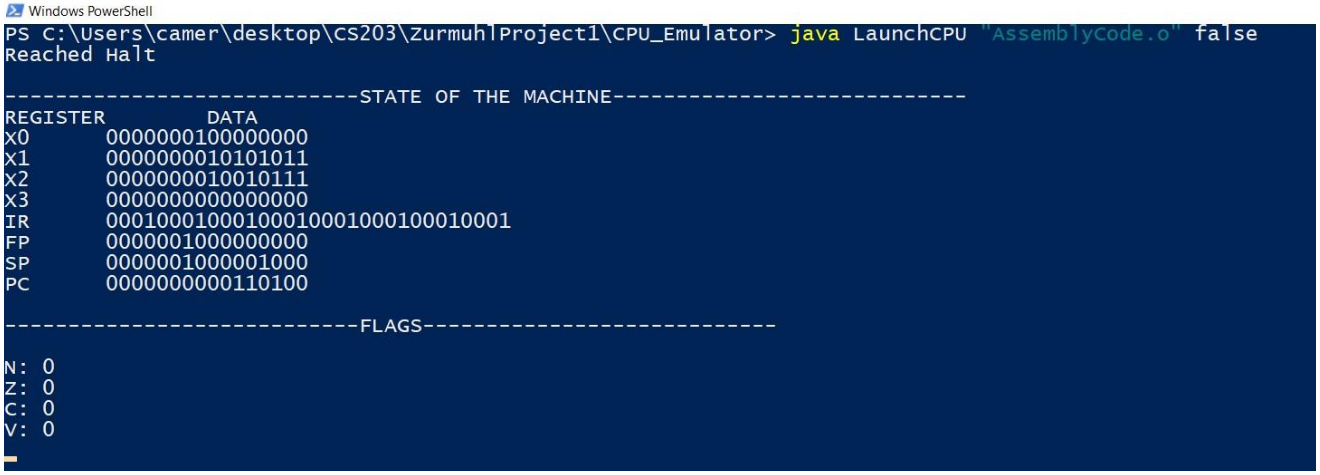


Figure 2: CPU Simulator GUI

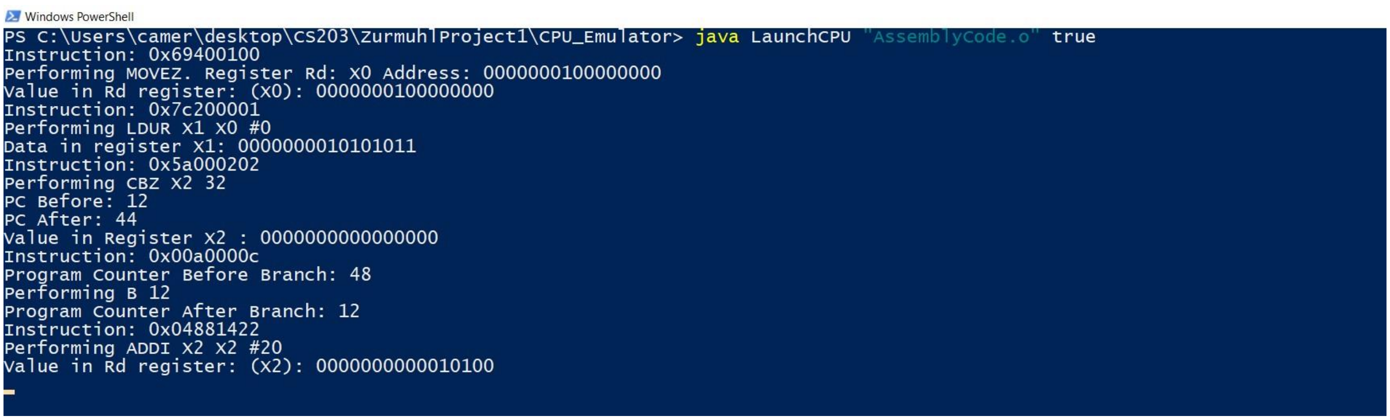
Here is an example state of the GUI the user can control. The left side of the GUI displays the information in the registers in binary. The right side of the GUI contains the controls, and the assembly instructions we are decoding. Starting with the Register Panel description—the time field describes which instruction the program is on. NZCV describes current flags. X0-IR are registers. The LEGv8 instruction is displayed for reference. The Control Panel allows user functionality to auto step through the number of instructions at three defined speeds. The user can reset the simulation to its initial state (everything contains zeros), exit the program, and export the current memory image to a new image-file, along with register states in another file. When the last command is reached (HALT), the user cannot advance the program, only reset. The state of the machine is printed to the command line:



```
Windows PowerShell
PS C:\Users\camer\desktop\CS203\ZurmuhlProject1\CPU_Emulator> java LaunchCPU "AssemblyCode.o" false
Reached Halt

----------------------------STATE OF THE MACHINE----------------------------
REGISTER        DATA
X0      0000000100000000
X1      0000000010101011
X2      0000000010010111
X3      0000000000000000
IR      00010001000100010001000100010001
FP      0000001000000000
SP      0000001000001000
PC      0000000000110100

----------------------------FLAGS----------------------------

N: 0
Z: 0
C: 0
V: 0
```

Figure 3: State of the Machine Example

Here is an example of noisy mode:



```
Windows PowerShell
PS C:\Users\camer\desktop\CS203\ZurmuhlProject1\CPU_Emulator> java LaunchCPU "AssemblyCode.o" true
Instruction: 0x69400100
Performing MOVEZ. Register Rd: X0 Address: 0000000100000000
Value in Rd register: (X0): 0000000100000000
Instruction: 0x7c200001
Performing LDUR X1 X0 #0
Data in register X1: 0000000010101011
Instruction: 0x5a000202
Performing CBZ X2 32
PC Before: 12
PC After: 44
Value in Register X2 : 0000000000000000
Instruction: 0x00a0000c
Program Counter Before Branch: 48
Performing B 12
Program Counter After Branch: 12
Instruction: 0x04881422
Performing ADDI X2 X2 #20
Value in Rd register: (X2): 0000000000010100
```

Figure 4: Noisy Mode

The other GUI is a memory viewer which displays the bounded memory. The user can look up any instruction by entering the hexadecimal address the instruction starts on. If the user enters an invalid input, the program handles it accordingly:

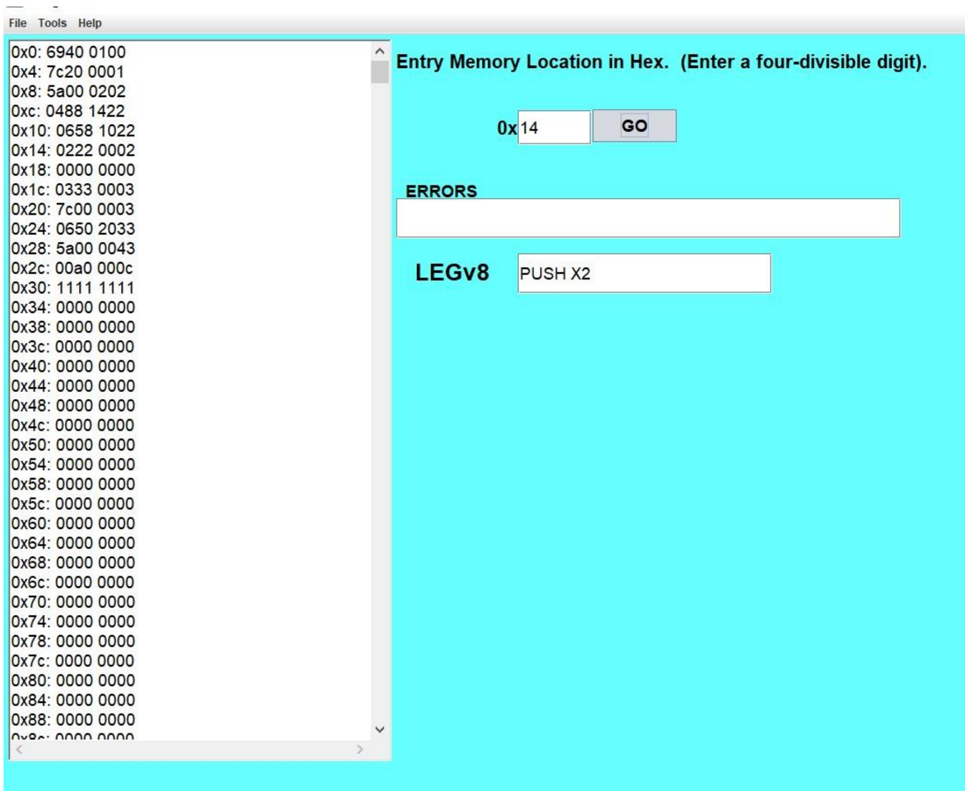


Figure 5: ImageVisualizer

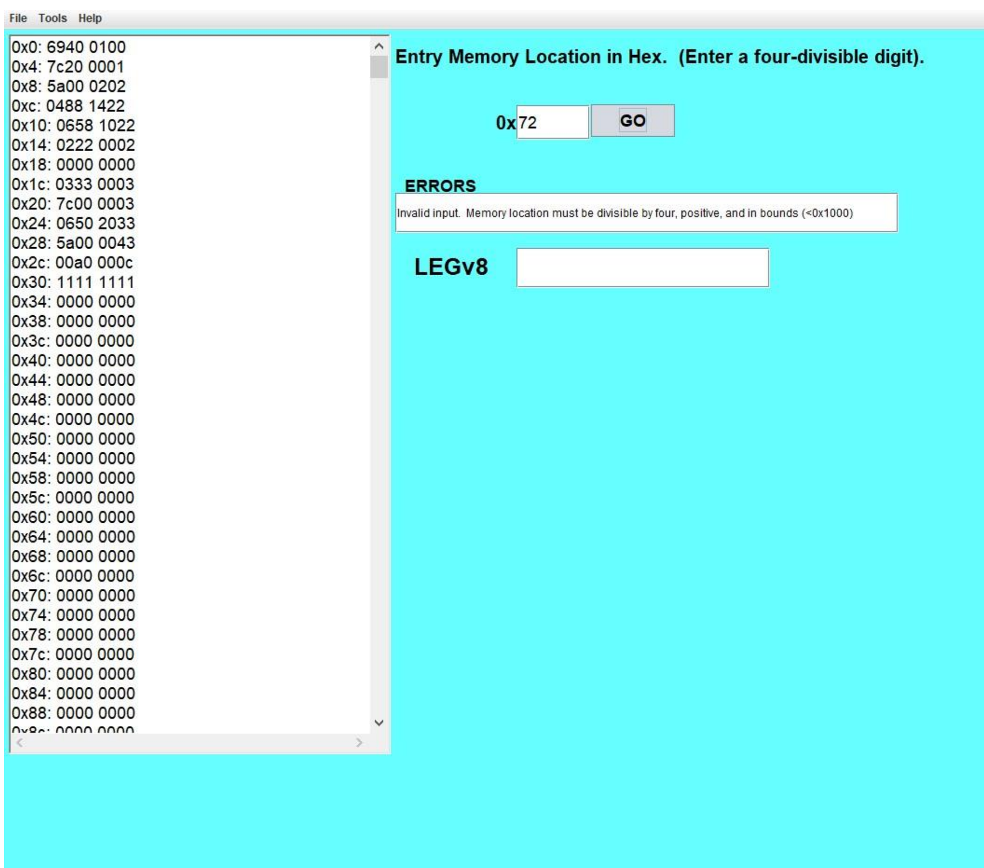


Figure 6: ImageVisualizer with error

Program 3: Viewer

The last part of the tool chain involved the ability to examine arbitrary image files at a specified range in either hexadecimal or binary. To change the implementation from the prior GUI, I limited what could be printed in the TextArea to include only a specified range (4). The same user functionalities cross apply from the ImageVisualzer GUI. There are restrictions on valid user input to the command line, which is specified in the program manual. The tool acts as a debugger by looking at smaller ranges of memory. Any image file can be passed to the viewer, and the specific command line arguments are discussed in the manual.

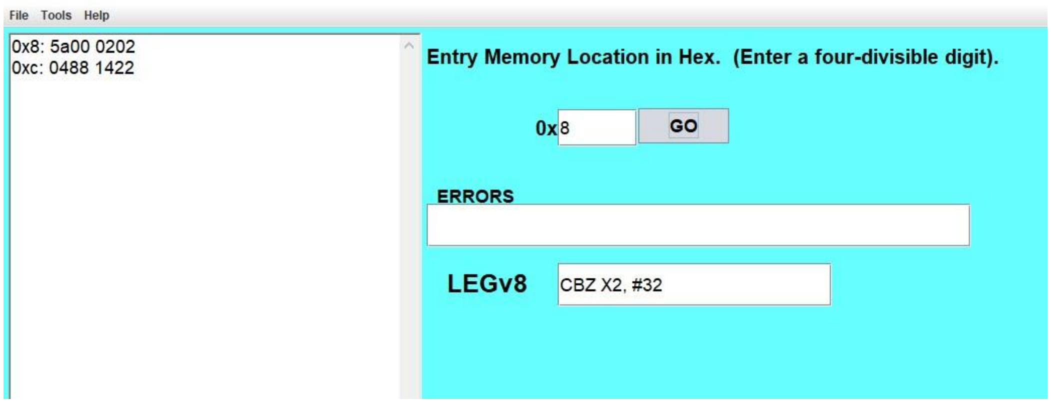


Figure 7: ImageVisualizer in Hex

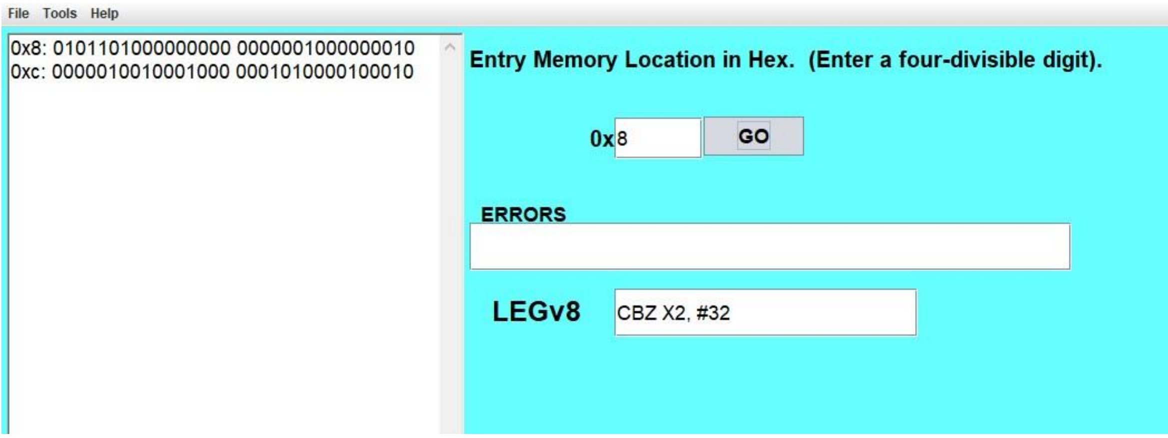


Figure 8: ImageVisualizer in Binary

Implementation of Instructions:

To implement the arithmetic, I used character arrays because of the input type being Strings. Any addition or subtraction was performed in an addStrings() method that added binary Strings formatted with a custom, static BinaryFormater class to the wordsize. The method would keep track of carry values during the process in a separate array, and set flags if necessary. Any data transfer type, like D types or MOVEZ, simply involved changing register data with setters. PUSH was implemented by adding a register's data to memory in a 2-byte field where the stack pointer starts. The SP advances two positions in memory, and the FP goes where the original SP was. POP acquires 2-bytes worth of memory starting where the FP points to, frees that memory, and brings the FP back 2 bytes. The SP then points to where the FP was originally pointing to. PUSH and POP operate with 2 bytes because that is the wordsize of the registers.

Reflection/Conclusion:

To date, this has been the most challenging computer science project I've completed. There were several new components and tools that I had to reason, like GUI programming and java makefiles. Once the design for the program was decided, the challenge became balancing time with how many instructions to implement. I had to make important design calls like which branching to use (PC relative), and what other instruction types to include. I add to ask questions like how was I going to store PUSH, POP, and MOVEZ as instruction types. Once these questions had answers, the rest of the programming fell in place. What I got out of this project is how to start thinking as a developer. Starting by modularizing the program with a design, discussing how to store abstract concepts discretely, and building relevant toolchains make the problem more approachable. In the end I was satisfied with my final product and now have a deeper understanding on the memory and control model.

References:

1. *Queue (Java Platform SE 8)*, Oracle, 8 Oct. 2017, docs.oracle.com/javase/8/docs/api/java/util/Queue.html.

2. *LinkedList (Java Platform SE 8)*, Oracle, 8 Oct. 2017, docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html.

3.*HashMap (Java Platform SE 8)*, Oracle, 8 Oct. 2017, docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

4. *TextArea (Java Platform SE 7)*, Oracle, 8 Oct. 2017, docs.oracle.com/javase/7/docs/api/java/awt/TextArea.html